

---

# Arrays- II

**CS10001: Programming & Data Structures**

**Pallab Dasgupta**  
**Dept. of Computer Sc. &**  
**Engg.,**  
**Indian Institute of**  
**Technology Kharagpur**

# Reading Array Elements

```
/* Read in student midterm and final grades and store them in two arrays*/
```

```
#define MaxStudents 100
```

```
int midterm[MaxStudents], final[MaxStudents];
```

```
int NumStudents ;    /* actual no of students */
```

```
int i, done, Smidterm, Sfinal;
```

```
printf ("Input no of students :");
```

```
scanf("%d", &NumStudents) ;
```

```
if (NumStudents > MaxStudents)
```

```
    printf ("Too many students") ;
```

```
else
```

```
    for (i=0; i<NumStudents; i++)
```

```
        scanf("%d%d", &midterm[i], &final[i]);
```

# Reading Arrays - II

```
/* Read in student midterm and final grades and store them in 2 arrays */
```

```
#define MaxStudents 100
```

```
int midterm[MaxStudents], final[MaxStudents];
```

```
int NumStudents ; /* actual no of students */
```

```
int i, done, Smidterm, Sfinal;
```

```
done=FALSE; NumStudents=0;
```

```
while (!done) {
```

```
    scanf("%d%d", &Smidterm, &Sfinal);
```

```
    if (Smidterm !=-1 || NumStudents>=MaxStudents)
```

```
        done = TRUE;
```

```
    else {
```

```
        midterm[NumStudents] = Smidterm;
```

```
        final[NumStudents] = Sfinal;
```

```
        NumStudents++;
```

```
    }
```

```
Dep }
```

# Size of an array

---

- How do you keep track of the number of elements in the array ?
  - 1. Use an integer to store the current size of the array.  
`#define MAX 100`  
`int size;`  
`float cost[MAX] ;`
  - 2. Use a special value to mark the last element in an array. If 10 values are stored, keep the values in `cost[0], ... , cost[9]`, have `cost[10] = -1`
  - 3. Use the 0th array element to store the size (`cost[0]`), and store the values in `cost[1], ... , cost[cost[0]]`

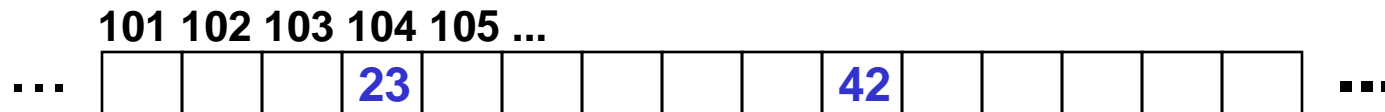
# Add an element to an array

---

1. `cost[size] = newval; size++;`
2. `for (i=0; cost[i] != -1; i++) ;`  
    `cost[i] = newval;`  
    `cost[i+1] = -1;`
3. `cost[0]++;`  
    `cost[cost[0]] = newval;`

# Address vs. Value

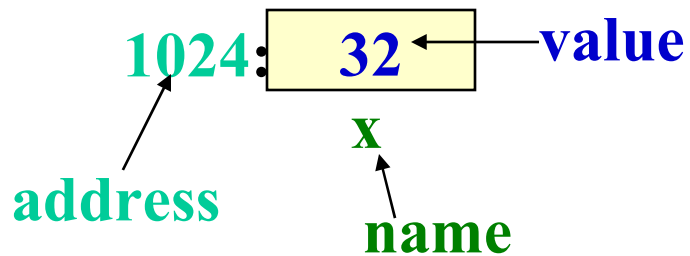
- Each memory cell has an **address** associated with it.
- Each cell also stores some **value**.
- Don't confuse the **address** referring to a memory location with the **value** stored in that location.





# Values vs Locations

- Variables name memory **locations**, which hold **values**.



**New Type : Pointer**

# Pointers

---

- A pointer is just a C variable whose value is the **address** of another variable!
- After declaring a pointer:

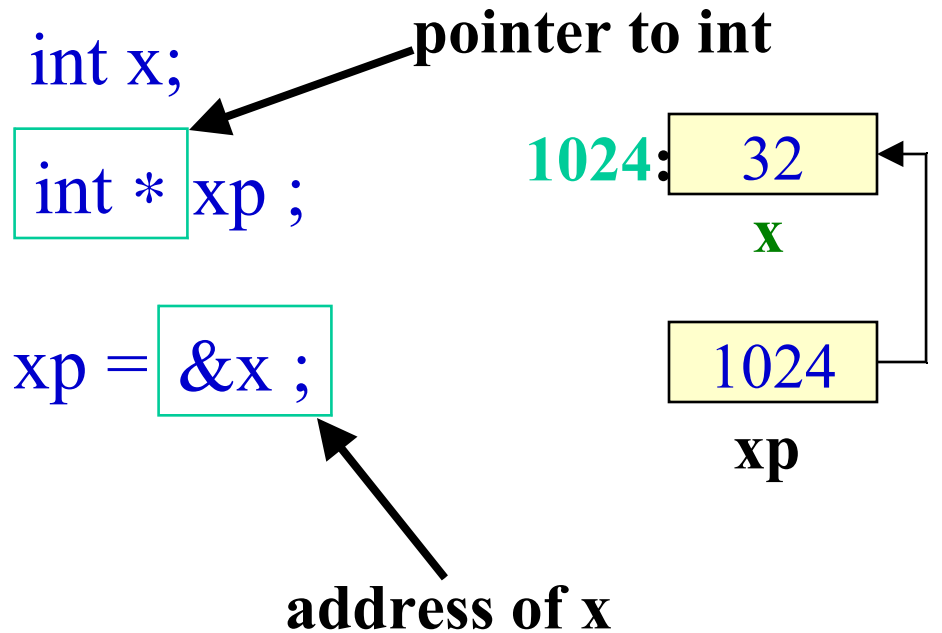
```
int *ptr;
```

`ptr` doesn't actually point to anything yet. We can either:

- make it point to something that already exists, or
- allocate room in memory for something new that it will point to... (next time)



# Pointer



```
*xp = 0;    /* Assign 0 to x */  
*xp = *xp + 1; /* Add 1 to x */
```

## Pointers Abstractly

```
int x;  
int * p;  
p=&x;  
...  
(x == *p)   True  
(p == &x)   True
```

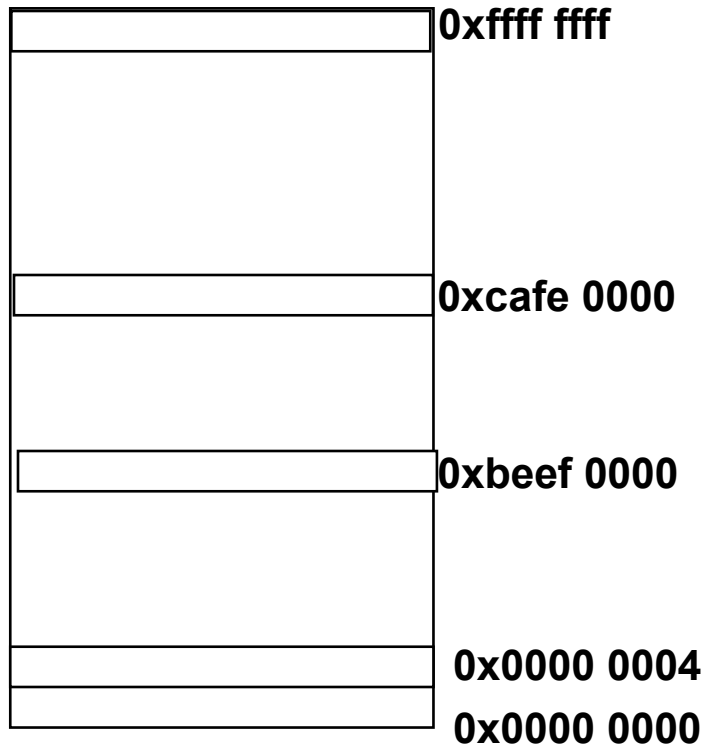
# Pointers

---

- **Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!**
- **Local variables in C are not initialized, they may contain anything.**

# Pointer Usage Example

## Memory and Pointers:



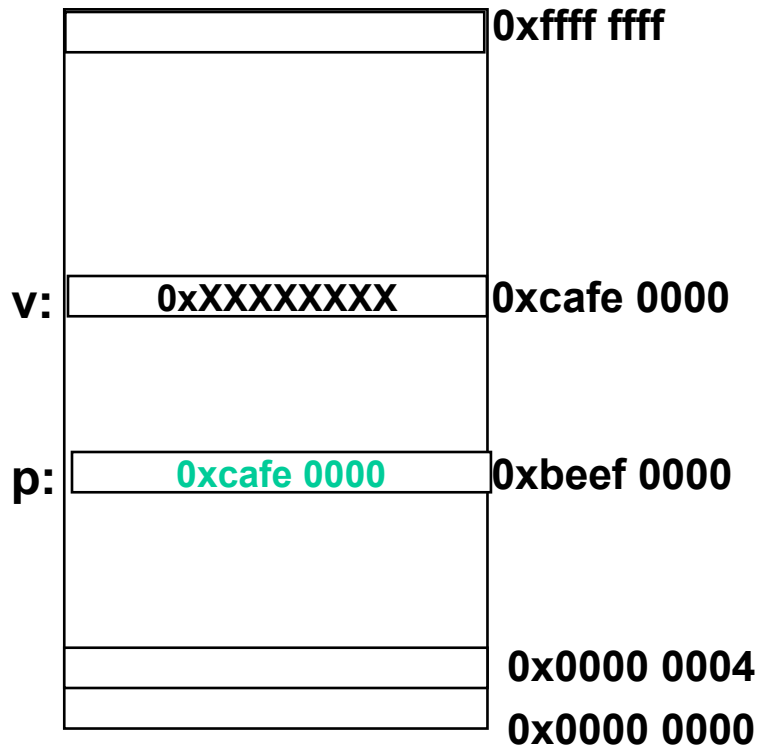
# Pointer Usage Example



Memory and Pointers:

```
int *p, v;
```

# Pointer Usage Example

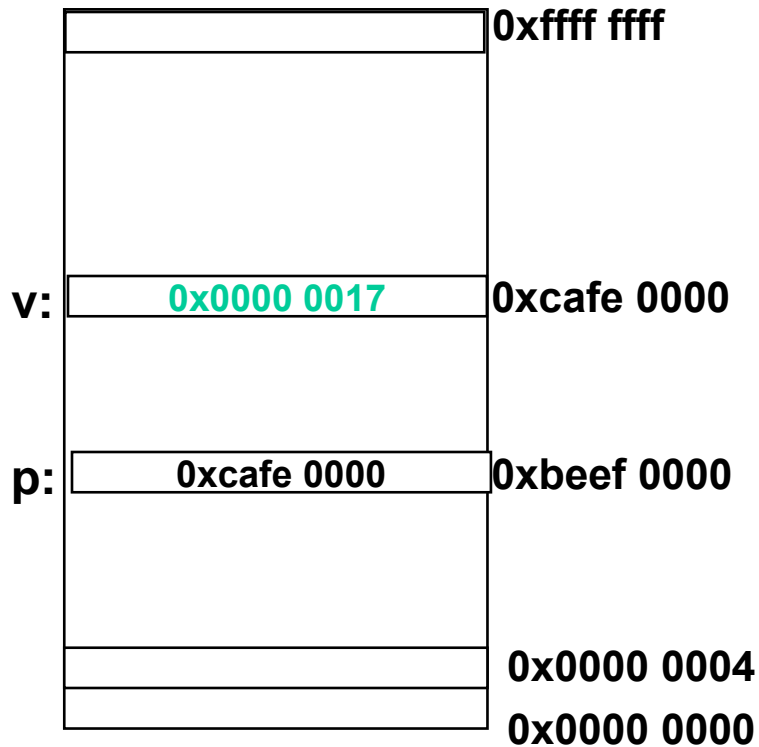


## Memory and Pointers:

```
int *p, v;
```

```
p = &v;
```

# Pointer Usage Example



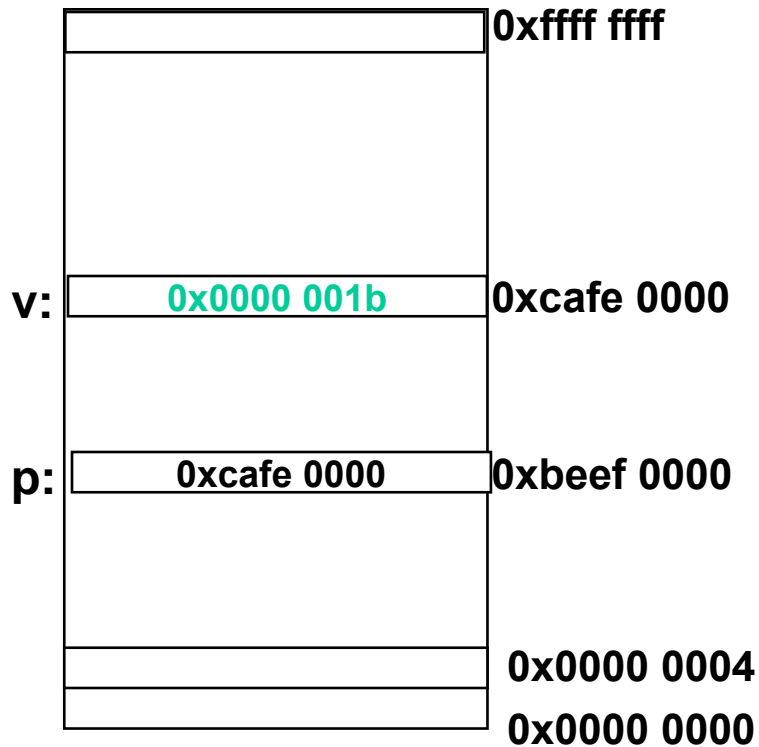
## Memory and Pointers:

```
int *p, v;
```

```
p = &v;
```

```
v = 0x17;
```

# Pointer Usage Example



## Memory and Pointers:

```
int *p, v;
```

```
p = &v;
```

```
v = 0x17;
```

```
*p = *p + 4;
```

```
V = *p + 4
```

# Arrays and pointers

---

- An array name is an address, or a pointer value.
- Pointers as well as arrays can be subscripted.
- A pointer variable can take different addresses as values.
- An array name is an address, or pointer, that is fixed. It is a **CONSTANT** pointer to the first element.



# Arrays

- **Consequences:**

- `ar` is a pointer
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`
- We can use pointer arithmetic to access arrays more conveniently.

- **Declared arrays are only allocated while the scope is valid**

```
char *foo() {  
    char string[32]; ...;  
    return string;  
} is incorrect
```

# Arrays

- Array size  $n$ ; want to access from 0 to  $n-1$ , so you should use counter AND utilize a constant for declaration & incr

– Wrong

```
int i, ar[10];  
for(i = 0; i < 10; i++){ ... }
```

– Right

```
#define ARRAY_SIZE 10  
int i, a[ARRAY_SIZE];  
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? SINGLE SOURCE OF TRUTH

– You're utilizing **indirection** and avoiding maintaining two copies of the number 10

# Arrays

- **Pitfall: An array in C does not know its own length, & bounds not checked!**
  - **Consequence: We can accidentally access off the end of an array.**
  - **Consequence: We must pass the array and its size to a procedure which is going to traverse it.**
- **Segmentation faults and bus errors:**
  - **These are VERY difficult to find; be careful!**
  - **You'll learn how to debug these in lab...**

# Arrays In Functions

---

- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.
  - Can be incremented

```
int strlen(char s[])  
{  
  
}
```

```
int strlen(char *s)  
{  
  
}
```

# Arrays and pointers

---

```
int a[20], i, *p;
```

- The expression **a[i]** is equivalent to **\*(a+i)**
- **p[i]** is equivalent to **\*(p+i)**
- When an array is declared the compiler allocates a sufficient amount of contiguous space in memory. The base address of the array is the address of **a[0]**.
- Suppose the system assigns 300 as the base address of **a**. **a[0]**, **a[1]**, ..., **a[19]** are allocated 300, 304, ..., 376.

# Arrays and pointers

```
#define N 20
```

```
int a[2N], i, *p, sum;
```

- `p = a;` is equivalent to `p = *a[0];`
- `p` is assigned 300.
- Pointer arithmetic provides an alternative to array indexing.
- `p=a+1;` is equivalent to `p=&a[1];` (`p` is assigned 304)

```
for (p=a; p<&a[N]; ++p)  
    sum += *p ;
```

```
for (i=0; i<N; ++i)  
    sum += *(a+i) ;
```

```
p=a;  
for (i=0; i<N; ++i)  
    sum += p[i] ;
```

# Arrays and pointers

---

```
int a[N];
```

- **a** is a **constant pointer**.
- ~~**a=p; ++a; a+=2; illegal**~~

# Pointer arithmetic and element size

---

```
double * p, *q ;
```

- The expression `p+1` yields the correct machine address for the next variable of that type.
- Other valid pointer expressions:
  - `p+i`
  - `++p`
  - `p+=i`
  - `p-q` /\* No of array elements between p and q \*/



# Arrays as parameters of functions

---

- **An array passed as a parameter is not copied**
- **An array name is a constant whose value serves as a reference to the first (index 0) item in the array.**

# Arrays as parameters of functions

---

- **Since constants cannot be changed, assignments to array variables are illegal.**
- **Only the array name is passed as the value of a parameter, but the name can be used to change the array's contents.**
- **Empty brackets [] are used to indicate that the parameter is an array. The no of elements allocated for the storage associated with the array parameter does not need to be part of the array parameter.**

# Pointer Arithmetic

- Since a pointer is just a mem address, we can add to it to traverse an array.
- $p+1$  returns a ptr to the next array elt.
- $(*p)+1$  vs  $*p++$  vs  $*(p+1)$  vs  $*(p)++$  ?
  - $x = *p++ \Rightarrow x = *p ; p = p + 1;$
  - $x = (*p)++ \Rightarrow x = *p ; *p = *p + 1;$
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality,  $p+1$  doesn't add 1 to the memory address, it adds the size of the array element.

# Pointer Arithmetic

---

- We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

- C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a char, 4 bytes for an int, etc.)

# Pointer Arithmetic

---

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
- So the following are equivalent:

```
int get(int array[], int n)
{
    return (array[n]);
    /* OR */
    return *(array + n);
}
```

# Pointer Arithmetic

- **Array size  $n$ ; want to access from 0 to  $n-1$** 
  - test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = ar; q = &(ar[10]);
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

- Is this legal?
- **C defines that one element past end of array must be a valid address, i.e., not cause an bus error or address error**

# Array operations

---

```
#define MAXS 100
int insert (int[], int, int, int) ;
int delete (int[], int, int) ;
int getelement (int[], int, int) ;
int readarray (int[], int) ;
main () {
    int a[MAXS];
    int size;
    size = readarray (a, 10) ;
    size = insert (a, size, 4, 7) ;
    x = getelement (a, size, 3) ;
    size = delete (a, size, 3) ;
}
```

# Array operations

```
#define MAXS 100
int insert (int[], int, int, int) ;
int delete (int[], int, int) ;
int getelement (int[], int, int) ;
int readarray (int[], int) ;
main () {
    int a[MAXS];
    int size;
    size = readarray (a, 10) ;
    size = insert (a, size, 4, 7) ;
    x = getelement (a, size, 3) ;
    size = delete (a, size, 3) ;
}
```

```
int readarray (int x[], int size) {
    int i;
    for (i=0; i<size; i++)
        scanf("%d", &x[i]) ;
    return size;
}
```

```
int getelement (int x[], int size, int pos)
if (pos <size) return x[pos] ;
return -1;
}
```

```
int insert (int x[], int size, int pos, int val) {
    for (k=size; k>pos; k--)
        x[k] = x[k-1] ;
    x[pos] = val ;
    return size+1;
}
```



---

```
void reverse (int x[],  
             int size) {
```

```
}
```

```
int findmax (int x[], int size)  
            {
```

```
}
```

---

```
void reverse (int x[], int size) {  
    int i;  
    for (i=0; i< (size/2); i++)  
        temp = x[size-i-1] ;  
        x[size-1-i] = x[i] ;  
        x[i] = temp;  
}
```

```
int findmax (int x[], int size) {  
    int i, max;  
    max = x[0];  
    for (i=1; i< size; i++)  
        if (x[i] > max)  
            max = x[i] ;  
    return max;  
}
```

# Two Dimensional Arrays

- We have seen that an array variable can store a list of values.
- Many applications require us to store a **table** of values.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Student 1	75	82	90	65	76
Student 2	68	75	80	70	72
Student 3	88	74	85	76	80
Student 4	50	65	68	40	70

## Contd.

---

- The table contains a total of 20 values, five in each line.
  - The table can be regarded as a **matrix** consisting of **four rows** and **five columns**.
- C allows us to define such tables of items by using **two-dimensional** arrays.

# Declaring 2-D Arrays

---

- **General form:**

```
type array_name [row_size][column_size];
```

- **Examples:**

```
int marks[4][5];
```

```
float sales[12][25];
```

```
double matrix[100][100];
```

# Accessing Elements of a 2-D Array

---

- **Similar to that for 1-D array, but use two indices.**
  - First indicates row, second indicates column.
  - Both the indices should be expressions which evaluate to integer values.

- **Examples:**

```
x[m][n] = 0;
```

```
c[i][k] += a[i][j] * b[j][k];
```

```
a = sqrt (a[j*3][k]);
```

# How is a 2-D array is stored in memory?

- Starting from a given memory location, the elements are stored **row-wise** in consecutive memory locations.
  - **x**: starting address of the array in memory
  - **c**: number of columns
  - **k**: number of bytes allocated per array element
- **a[i][j]** → is allocated memory location at address  **$x + (i * c + j) * k$**

a[0][0] a[0][1] a[0][2] a[0][3] a[1][0] a[1][1] a[1][2] a[1][3] a[2][0] a[2][1] a[2][2] a[2][3]

Row 0

Row 1

Row 2

# How to read the elements of a 2-D array?

---

- **By reading them one element at a time**

```
for (i=0; i<nrow; i++)  
    for (j=0; j<ncol; j++)  
        scanf ("%f", &a[i][j]);
```
- **The ampersand (&) is necessary.**
- **The elements can be entered all in one line or in different lines.**



# How to print the elements of a 2-D array?

---

- **By printing them one element at a time.**

```
for (i=0; i<nrow; i++)  
    for (j=0; j<ncol; j++)  
        printf (“\n %f”, a[i][j]);
```

- **The elements are printed one per line.**

```
for (i=0; i<nrow; i++)  
    for (j=0; j<ncol; j++)  
        printf (“%f”, a[i][j]);
```

- **The elements are all printed on the same line.**

## Contd.

---

```
for (i=0; i<nrow; i++)
{
    printf ("\n");
    for (j=0; j<ncol; j++)
        printf ("%f  ", a[i][j]);
}
```

- The elements are printed nicely in matrix form.

## Example: Matrix Addition

```
#include <stdio.h>
int main() {
    int a[100][100], b[100][100],
        c[100][100], p, q, m, n;

    scanf ("%d %d", &m, &n);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &a[p][q]);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &b[p][q]);
```

```
    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            c[p][q] = a[p][q] + b[p][q];

    for (p=0; p<m; p++) {
        printf ("\n");
        for (q=0; q<n; q++)
            printf ("%f  ", a[p][q]);
    }
}
```

# Passing Arrays to Function

---

- Array element can be passed to functions as ordinary arguments.
  - `IsFactor (x[i], x[0])`
  - `sin (x[5])`

# Passing Entire Array to a Function

---

- **An array name can be used as an argument to a function.**
  - Permits the entire array to be passed to the function.
  - The way it is passed differs from that for ordinary variables.
- **Rules:**
  - The array name must appear by itself as argument, without brackets or subscripts.
  - The corresponding formal argument is written in the same manner.
    - Declared by writing the array name with a pair of empty brackets.

# Whole array as Parameters

```
#define ASIZE 5
float average (int a[])      {
    int i, total=0;
    for (i=0; i<ASIZE; i++)
        total = total + a[i];
    return ((float) total / (float) ASIZE);
}

main ( ) {
    int x[ASIZE] ; float x_avg;
    x = {10, 20, 30, 40, 50}
    x_avg = average (x) ;
}
```

## Contd.

We don't need to write the array size. It works with arrays of any size.

```
int main()
{
    int n;
    float list[100], avg;
    :
    avg = average (n, list);
    :
}

float average (a, x)
int a;
float x[];
{
    :
    sum = sum + x[i];
}
```

# Arrays as Output Parameters

```
void VectorSum (int a[], int b[], int vsum[], int length)    {
    int i;
    for (i=0; i<length; i=i+1)
        vsum[i] = a[i] + b[i] ;
}
int main (void)      {
    int x[3] = {1,2,3}, y[3] = {4,5,6}, z[3];
    VectorSum (x, y, z, 3) ;
    PrintVector (z, 3) ;
}
void PrintVector (int a[], int length)    {
    int i;
    for (i=0; i<length; i++) printf ("%d ", a[i]);
}
```



# The Actual Mechanism

---

- When an array is passed to a function, the values of the array elements are **not passed** to the function.
  - The array name is interpreted as the **address** of the first array element.
  - The formal argument therefore becomes a **pointer** to the first array element.
  - When an array element is accessed inside the function, the address is calculated using the formula stated before.
  - **Changes made inside the function are thus also reflected in the calling program.**

## Contd.

---

- Passing parameters in this way is called **call-by-reference.**
- Normally parameters are passed in C using **call-by-value.**
- **Basically what it means?**
  - If a function changes the values of array elements, then these changes will be made to the original array that is passed to the function.
  - This does not apply when an individual element is passed on as argument.

# Passing 2-D Arrays

---

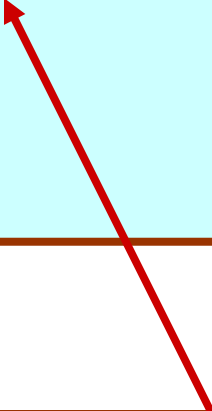
- **Similar to that for 1-D arrays.**
  - The array contents are not copied into the function.
  - Rather, the address of the first element is passed.
- **For calculating the address of an element in a 2-D array, we need:**
  - The starting address of the array in memory.
  - Number of bytes per element.
  - Number of columns in the array.
- **The above three pieces of information must be known to the function.**

# Example Usage

```
#include <stdio.h>

main()
{
    int a[15][25], b[15][25];
    :
    :
    add (a, b, 15, 25);
    :
}
```

```
void add (x, y, rows, cols)
int x[][25], y[][25];
int rows, cols;
{
    :
}
```



We can also write

```
int x[15][25], y[15][25];
```